R Small Group: Class 6

Amy Allen & Dayne Filer

July 19, 2016

Using this document

- Code blocks and R code have a grey background (note, code nested in the text is not highlighted in the pdf version of this document but is a different font).
- # indicates a comment, and anything after a comment will not be evaluated in R
- The comments beginning with ## under the code in the grey code boxes are the output from the code directly above; any comments added by us will start with a single #
- While you can copy and paste code into R, you will learn faster if you type out the commands yourself.
- Read through the document after class. This is meant to be a reference, and ideally, you should be able to understand every line of code. If there is something you do not understand please email us with questions or ask in the following class (you're probably not the only one with the same question!).

Class 6 expectations

1. Know how to install and load R packages
2. Consider some principles of reproducible research
3. Know the basic components of an R package
4. Create a simple R package using RStudio and roxygen2

Introduction to R packages

So far in this class you have only used functions that are included within the basic installation of R. R, being an open source language, allows any user to write their own additions to R formally called "extensions," but more casually (and commonly) referred to as "packages."

While there are many repositories for R packages online, the primary repository is called "The Comprehensive R Archive Network" or "CRAN." By default, R will search for packages on CRAN. You can install a new package using the `install.packages` function:

```r
install.packages("VennDiagram", repos = "http://archive.linux.duke.edu/cran/")
##
## The downloaded binary packages are in
##  /var/folders/yb/9clrgrjd3_j7d15sf25n05300000gn/T//Rtmp5jE1sz/downloaded_packages
```

Note, you do not have to set `repos` in the code above. If you omit `repos` R will ask you to select a "mirror" from a list. Copies of CRAN are hosted on many servers around the world, and it is suggested that you use the closest host for installing packages. Once you make your selection, R will remember your selection for the remainder of that session. When working in RStudio, the mirror will automatically be set to 'https://cran.rstudio.com.'

Now that the `VennDiagram` package is installed, you can use the `library` function to load the package into R.

```r
library(VennDiagram)
## Loading required package: grid
## Loading required package: futile.logger
```

Note, when you run `ls` you do not see the contents of the package. You do not need to understand the complexities of how R works, but briefly, there are other environments that sit above the global environment. Look at the output from `search`:

```
search()
##  [1] ".GlobalEnv"            "package:VennDiagram"
##  [3] "package:futile.logger" "package:grid"
##  [5] "package:stats"         "package:graphics"
##  [7] "package:grDevices"     "package:utils"
##  [9] "package:datasets"      "package:methods"
## [11] "Autoloads"             "package:base"
```

Here, you see you have been using packages all along. For example, the `plot` function is part of the `graphics` package. Notice the `VennDiagram` package in your search path. If you wish to "unload" a package (remove it from your search path), use the `unloadNamespace` function:

```
unloadNamespace("VennDiagram")
```

Look at the output from the original from when you loaded `VennDiagram` with `library`, and notice R told you two other required packages were loaded. You would also have to remove `grid` and `futile.logger`:

```
search()
##  [1] ".GlobalEnv"            "package:futile.logger"
##  [3] "package:grid"          "package:stats"
##  [5] "package:graphics"      "package:grDevices"
##  [7] "package:utils"         "package:datasets"
##  [9] "package:methods"       "Autoloads"
## [11] "package:base"
unloadNamespace("grid")
## Warning: shutting down all devices when unloading 'grid' namespace
unloadNamespace("futile.logger")
search()
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

The abundance and diversity of R packages are what make R a popular choice for computational research – as you begin to do your research chances are someone has already solved the problem (or a very similar problem), and you can take advantage of preexisting code.

Some tenets of reproducible research

As with any research, computational work should be reproducible. Unfortunately, much of the computational research being done now is difficult or impossible to reproduce. While this is not meant to be the definitive guide on reproducible computational research, there are some general ideas you should be aware of and some simple steps you can take to increase the reproducibility (and, therefore, the value) of your research:

- Use strict syntax standards
- Make ***thorough*** comments throughout your code
- Include documentation with all code you write
- Consider how another scientist might run your code within his or her environment
- Use version control software (VCS) to track your work

A few words on readability

One software developer said:

> Always code as if the guy who ends up maintaining [or trying to understand] your code will be a violent psychopath who knows where you live. Code for readability.

The two most important things you can do to increase the readability of your code are: (1) be relentlessly judicious with your coding syntax and (2) comment often. Google provides an excellent style guide that, if followed, will make your code easy to read. You can find the style guide at: https://google.github.io/styleguide/Rguide.

You should also write frequent comments explaining the justification for the code. Most languages have a comment character. In R, the pound sign (`#`) is the comment character. Anything after a pound sign (until the next new line) will not be evaluated when the code is run. Consider the code below:

```r
var1 <- c(1:5)
var2 <- mean(var1)*1023
```

Consider the following two comments for the code above:

```r
## Assign 1:5 to var1, then take the mean of var1 and multiply by 1023
var1 <- c(1:5)
var2 <- mean(var1)*1023

## From Filer et al. 2016, we want to multiply the mean of the results by 1023
## to get the correct estimate of the parameter
var1 <- c(1:5)
var2 <- mean(var1)*1023

## Remove var1 and var2 so the global environment is clean again
rm(var1, var2)
```

This code needs a comment to explain *why*, not *what*, is happening. You can assume the reader would know that you took the mean of `var1` and multiplied by `1023` by looking at the code. Comments should most often explain the justification for the code, not the workings of the code (unless the code is complicated such that you think readers might not understand, in which case you would want to explain why *and* how).

Do not confuse commenting code with documentation. Code should have comments that explain the code, how it works and what it does. Documentation should explain how to use the program, and incude every piece of information someone would need to successfully run the program on their own. Well commented code does not substitute for documentation, nor does documentation substitute for well commented code. Both well commented code and documentation are necessary for clear reproducible research.

Again, be consistent and clear in your syntax and always add appropriate comments. You will certainly thank yourself for doing so later.

Version control

Version control is a crucial aspect of computational research, and, unfortunately, outside the scope of this class. You are encouraged to use git and GitHub. There are many guides online to show you how to get started with git, including some great guides on GitHub. If you want to "get" serious about git you should consider reading *Pro Git*: https://git-scm.com/book/en/v2

An argument for the R package

Conventionally, R packages contain a group of functions and data structures to extend the functionality in R. For example, the `VennDiagram` function installed above creates Venn diagrams – a functionality not included in the base distribution of R. In addition to storing functions, R packages include all the documentation, and can also include data. Therefore, we argue R packages provide an ideal environment for producing and distributing reproducible research.

After learning the components of an R package, you will create a simple R package to illustrate how you would package up your own research project, complete with data and documentation.

Components of an R package

All R packages have the same basic strcture:

- DESCRIPTION – a file with the meta data about your package, the name, authors, version, etc.
- R – a folder containing .R files that define the functions for your package
- man – a folder with .Rmd fils that document your functions
- NAMESPACE – a file defining which functions are visible to the user, and which functions from other packaages your package uses
- vignettes – a folder containing the (optional) vignettes for your package

You can think of a vignette as help file for the package as a whole, explaining what the package does (in more detail than the DESCRIPTION file). We will not discuss how to write vignettes (there are multiple good options, with varrying complexity and capability). It is worth noting that some researchers are starting to write their manuscripts corresponding to a package as a vignette, making the R package truly self-contained.

You can look up how these files are formatted (R provides the extensive "Writing R Extensions" website, https://cran.r-project.org/doc/manuals/R-exts.html, with all the neccessary details), and generate them yourself. Or, you can let RStudio and the `roxygen2` package to do the hard work for you.

Create a package with RStudio and roxygen2

There is a great and short video on YouTube showing the following steps: https://www.youtube.com/watch?v=9PyQlbAEujY.

First, in the RStudio menu bar go to File -> New Project. Then select "New Directory" when pop-up appears. Select "R Package" and enter "firstpackage" as the R package name. At this step you could also select R files you have already written, and change where the new folder is created.

At this point a new RStudio window should open with package structure established. The new folder has two additional files: (1) firstpackage.Rproj and (2) .Rbuildignore. The fristpackage.Rproj file is an RStudio file that contains information about how RStudio managaes the project for you. The .Rbuildignore file tells R which files to ignore when creating the package, i.e. the firstpackage.Rproj file.

The next step is to edit the DESCRIPTION file. Fill out your DESCRIPTION file:

```
Package: firstpackage
Type: Package
Title: My first package
Version: 0.1.0
Author: Amy & Dayne
Maintainer: Dayne Filer <dayne_filer@med.unc.edu>
Description: This package has one function and was created to show how to
```

```
build a package.
License: GPL-2
LazyData: TRUE
```

Then go into the R folder and open the hello.R file. This file shows how you add functions to your package.

Next, install the `roxygen2` package and configure RStudio to use `roxygen`. On the menu bar go to Build -> Configure Build Tools. When the popup menu appears click on checkbox for "Generate documentation with Roxygen" and check all the boxes.

The `roxygen2` package provides a simple syntax for integrating the documentation into the R files. For every R function you write you should include an Roxygen header at the top of the file. Roxygen headers begin with `#'`. You can then specify the sections of the help files with the `@` sign. Minimally, R help files require a title, description, and the parameters. Using Roxygen, you can also specify whether to export the function with the `@export` tag. (If you do not add the `@export` tag, the user will not be able to see that function when your package is loaded.)

Open a new R script (Menu -> File -> New File -> R Script). Type (or copy) the following code:

```
#' @title Return the input
#' @description This function simply returns the input, 'x'
#' @param x An R object
#' @export

returnInput <- function(x) {
  return(x)
}
```

Now delete the R/hello.R and man/hello.Rmd files. Finally, build the package by clicking on Menu -> Build -> Build and Reload. After the package builds, you should see `library(firstpackage)` run in your console and you are ready to use your package. Try typing the following:

```r
search()
##  [1] ".GlobalEnv"           "package:firstpackage" "package:stats"
##  [4] "package:graphics"     "package:grDevices"    "package:utils"
##  [7] "package:datasets"     "package:methods"      "Autoloads"
## [10] "package:base"
returnInput("hello")
## [1] "hello"
?returnInput
```

As you can see your package is loaded into your search path, you can use your function, and your function has a help file. In the same manner you can add more functions to your package.

Add data to your package

Finally, you will add some data to your package. First create a new folder called "data" in your "firstpackage" folder. Then create an object, `mydata` and use the `save` function to save it to the "data" folder you just created.

```r
mydata <- matrix(1:25, ncol = 5)
save(mydata, file = file.path("data", "mydata.RData"))
rm(mydata)
```

Now run Build and Install again. After loading the pacakge you can load `mydata` into your environment with the `data` function:

```
ls()
## character(0)
data("mydata")
mydata
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

You can (and should) document your datasets as well. Create a new R file in the "R" directory called "mydata.R" and add/copy the following text:

```
#' @name mydata
#' @title Dataset for firstpackage
#' @description This is an example dataset to illustrate how to add data to packages
#' @details Not much to say here

NULL
```

Build and reload again, then look at the help file for `mydata`:

```
?mydata
```

Class 6 excercises

1. Create an R package called "fruitmachine."
2. Add the `genFruit` and `fruit_machine` functions from class 5 and document them.
3. Add the `farmer1` and `farmer2` objects to the package and document them.